Artificial Intelligence lab (18CS680)

by Nitin Vinayak

```
# BFS & DFS Traversal

# BFS : Breadth First Search
# DFS : Depth First Search

print("Enter the Number of Nodes : ")
n = int(input())
nodes = [[0 for i in range(n+1)] for i in range(n+1)]
visited = [0 for i in range(n+1)]
n = n + 1
print("Enter the Number of Edges : ")
edges = int(input())
print("Enter the Edges (a b) : ")
for i in range(edges):
  a,b = map(int,input().split())
  nodes[a][b] = 1
  nodes[b][a] = 1
print("Enter the Starting Element : ")
start = int(input())

def bfs(x):
  print("BFS : ", end = '')
  queue = [x]
  while(len(queue) != 0):
    current_node = queue[0]
    visited[current_node] = 1
    queue = queue[1:]
    print(current_node,end = ' ')
    for i in range(n):
      if(visited[i] == 0 and nodes[current_node][i] == 1):
        queue.append(i)
bfs(start)
print()
visited = [0 for i in range(n)]

def dfs(x):
  print(x,end = ' ')
  visited[x] = 1
  for i in range(n):
    if(nodes[x][i] == 1 and visited[i] == 0):
      dfs(i)

print("DFS : ", end = '')
dfs(start)
```

```
Enter the Number of Nodes :
```

```
# BFS Traversal for the given Viva Qn
```

```
graph = {
  'A':['B','C'],
  'B':['D','E'],
  'C':['F','G'],
  'D':['H','I'],
  'E':['J','K'],
  'F':['L','M'],
  'G':['N','O'],
  'H':['I'],
  'I':['J'],
  'J':['K'],
  'K':['L'],
  'L':['M'],
  'M':['N'],
  'N':['O'],
  'O':[]
}

visited = []
queue = []

def bfs(visited, graph, node,ending):
  visited.append(node)
  queue.append(node)

  while queue:
    m = queue.pop(0)
    print (m, end = " ")
    if(m == ending):
      break

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

print("Using BFS : ")
bfs(visited, graph, 'A','H')
```

```
Using BFS :
A B C D E F G H
```

```
# DFS Traversal for the given Viva Qn

graph = {
  'A':['B','C'],
  'B':['D','E'],
  'C':['F','G'],
  'D':['H','I'],
  'E':['J','K'],
  'F':['L','M'],
  'G':['N','O'],
  'H':['I'],
```

```
  'I':['J'],
  'J':['K'],
  'K':['L'],
  'L':['M'],
  'M':['N'],
  'N':['O'],
  'O':[]
}

visited = set()

def dfs(visited, graph, node):
    if node not in visited:
        print (node, end = ' ')
        visited.add(node)
        if(node == 'H'):
          return
        for neighbour in graph[node]:
            return dfs(visited, graph, neighbour)

print("Using DFS : ")
dfs(visited, graph, 'A')
```

```
Using DFS :
A B D H
```

```
# Water Jug Problem

from collections import deque

def BFS(a, b, target):
    m = {}
    isSolvable = False
    path = []
    q = deque()
    q.append((0, 0))

    while (len(q) > 0):
        u = q.popleft()

        if ((u[0], u[1]) in m):
            continue

        if ((u[0] > a or u[1] > b or
            u[0] < 0 or u[1] < 0)):
            continue

        path.append([u[0], u[1]])
        m[(u[0], u[1])] = 1

        if (u[0] == target or u[1] == target):
            isSolvable = True

            if (u[0] == target):
```

```
                if (u[1] != 0):
                    path.append([u[0], 0])
            else:
                if (u[0] != 0):
                    path.append([0, u[1]])

            sz = len(path)
            for i in range(sz):
                print("(", path[i][0], ",",
                        path[i][1], ")")
            break

        q.append([u[0], b]) # Fill Jug2
        q.append([a, u[1]]) # Fill Jug1

        for ap in range(max(a, b) + 1):

            c = u[0] + ap
            d = u[1] - ap

            if (c == a or (d == 0 and d >= 0)):
                q.append([c, d])

            c = u[0] - ap
            d = u[1] + ap

            if ((c == 0 and c >= 0) or d == b):
                q.append([c, d])

        q.append([a, 0])
        q.append([0, b])

    if (not isSolvable):
        print ("No solution")

if __name__ == '__main__':

    Jug1, Jug2, target = 4, 3, 2
    print("Path from initial state "
        "to solution state ::")

    BFS(Jug1, Jug2, target)
```

```
Path from initial state to solution state ::
( 0 , 0 )
( 0 , 3 )
( 4 , 0 )
( 4 , 3 )
( 3 , 0 )
( 1 , 3 )
( 3 , 3 )
( 4 , 2 )
( 0 , 2 )
```

```
# N Queens Problem

from queue import Queue

class NQueens:

    def __init__(self, size):
        self.size = size

    def solve_dfs(self):
        if self.size < 1:
            return []
        solutions = []
        stack = [[]]
        while stack:
            solution = stack.pop()
            if self.conflict(solution):
                continue
            row = len(solution)
            if row == self.size:
                solutions.append(solution)
                continue
            for col in range(self.size):
                queen = (row, col)
                queens = solution.copy()
                queens.append(queen)
                stack.append(queens)
        return solutions

    def solve_bfs(self):
        if self.size < 1:
            return []
        solutions = []
        queue = Queue()
        queue.put([])
        while not queue.empty():
            solution = queue.get()
            if self.conflict(solution):
                continue
            row = len(solution)
            if row == self.size:
                solutions.append(solution)
                continue
            for col in range(self.size):
                queen = (row, col)
                queens = solution.copy()
                queens.append(queen)
                queue.put(queens)
        return solutions

    def conflict(self, queens):
        for i in range(1, len(queens)):
            for j in range(0, i):
                a, b = queens[i]
                c, d = queens[j]
```

```
                if a == c or b == d or abs(a - c) == abs(b - d):
                    return True
        return False

    def print(self, queens):
        for i in range(self.size):
            print(' ---' * self.size)
            for j in range(self.size):
                p = 'Q' if (i, j) in queens else ' '
                print('| %s ' % p, end='')
            print('|')
        print(' ---' * self.size)

def main():
    size = int(input('Enter the Size of the Chess Board : '))
    print ('\n')
    n_queens = NQueens(size)
    dfs_solutions = n_queens.solve_dfs()
    bfs_solutions = n_queens.solve_bfs()
    for i, solution in enumerate(dfs_solutions):
        print('DFS Solution %d:' % (i + 1))
        n_queens.print(solution)
        print ('\n')
    for i, solution in enumerate(bfs_solutions):
        print('BFS Solution %d:' % (i + 1))
        n_queens.print(solution)
        print('\n')
    print('Total Number of Solutions using DFS : %d' % len(dfs_solutions))
    print('Total Number of Solutions using BFS : %d' % len(bfs_solutions))


if __name__ == '__main__':
    main()
```

```
Enter the Size of the Chess Board : 4


DFS Solution 1:
 --- --- --- ---
|   |   | Q |   |
 --- --- --- ---
| Q |   |   |   |
 --- --- --- ---
|   |   |   | Q |
 --- --- --- ---
|   | Q |   |   |
 --- --- --- ---


DFS Solution 2:
 --- --- --- ---
|   | Q |   |   |
 --- --- --- ---
|   |   |   | Q |
 --- --- --- ---
| Q |   |   |   |
 --- --- --- ---
```

```
|   |   | Q |   |
 --- --- --- ---


BFS Solution 1:
 --- --- --- ---
|   | Q |   |   |
 --- --- --- ---
|   |   |   | Q |
 --- --- --- ---
| Q |   |   |   |
 --- --- --- ---
|   |   | Q |   |
 --- --- --- ---


BFS Solution 2:
 --- --- --- ---
|   |   | Q |   |
 --- --- --- ---
| Q |   |   |   |
 --- --- --- ---
|   |   |   | Q |
 --- --- --- ---
|   | Q |   |   |
 --- --- --- ---


Total Number of Solutions using DFS : 2
Total Number of Solutions using BFS : 2
```

```
# Uniform Cost Search

def uniform_cost_search(goal, start):
    global graph,cost
    answer = []
    queue = []
    for i in range(len(goal)):
        answer.append(10**8)
    queue.append([0, start])
    visited = {}
    count = 0
    while (len(queue) > 0):
        queue = sorted(queue)
        p = queue[-1]
        del queue[-1]
        p[0] *= -1
        if (p[1] in goal):
            index = goal.index(p[1])
            if (answer[index] == 10**8):
                count += 1
            if (answer[index] > p[0]):
                answer[index] = p[0]
            del queue[-1]
            queue = sorted(queue)
            if (count == len(goal)):
```

```
                return answer
        if (p[1] not in visited):
            for i in range(len(graph[p[1]])):
                queue.append( [(p[0] + cost[(p[1], graph[p[1]][i])])* -1, graph[p[
        visited[p[1]] = 1
    return answer

if __name__ == '__main__':

    graph,cost = [[] for i in range(8)],{}

    graph[0].append(1)
    graph[0].append(3)
    graph[3].append(1)
    graph[3].append(6)
    graph[3].append(4)
    graph[1].append(6)
    graph[4].append(2)
    graph[4].append(5)
    graph[2].append(1)
    graph[5].append(2)
    graph[5].append(6)
    graph[6].append(4)

    cost[(0, 1)] = 2
    cost[(0, 3)] = 5
    cost[(1, 6)] = 1
    cost[(3, 1)] = 5
    cost[(3, 6)] = 6
    cost[(3, 4)] = 2
    cost[(2, 1)] = 4
    cost[(4, 2)] = 4
    cost[(4, 5)] = 3
    cost[(5, 2)] = 6
    cost[(5, 6)] = 3
    cost[(6, 4)] = 7

    goal = []
    goal.append(6)
    answer = uniform_cost_search(goal, 0)
    print("Minimum cost from 0 to 6 is = ", answer[0])
```

```
     Minimum cost from 0 to 6 is =  3
```

```
#Iterative Deepening Search

def iterative_deepening_dfs(start, target):
    depth = 1
    bottom_reached = False
    while not bottom_reached:
        result, bottom_reached = iterative_deepening_dfs_rec(start, target, 0, dep
        if result is not None:
            return result
        depth *= 2
        print("Increasing depth to " + str(depth))
```

```
        return None

    def iterative_deepening_dfs_rec(node, target, current_depth, max_depth):
        print("Visiting Node " + str(node["value"]))
        if node["value"] == target:
            print("Found the node we're looking for!")
            return node, True
        if current_depth == max_depth:
            print("Current maximum depth reached, returning...")
            if len(node["children"]) > 0:
                return None, False
            else:
                return None, True
        bottom_reached = True
        for i in range(len(node["children"])):
            result, bottom_reached_rec = iterative_deepening_dfs_rec(node["children"][
            if result is not None:
                return result, True
            bottom_reached = bottom_reached and bottom_reached_rec
        return None, bottom_reached



    # 8 Puzzle problem using Iterative Deepening Search

    def printState_8p(state):
        ctr = 0
        for i in range(3):
            for j in range(3):
                if state[ctr] == 0:
                    print(' ', end = ' ')
                else:
                    print(state[ctr], end=' ')
                ctr += 1
            print()

    def printPath_8p(startState, goalState, path):
        l = len(path)
        print("The path from %s to %s is %d nodes long." % (startState, goalState, l))
        print()
        print(type(path))
        print()
        for p in path:
            printState_8p(p)
            print()

    def matrix_to_list(x, y):
        counter = 0
        for i in range(3):
            for j in range(3):
                if i == x and j == y:
                    return counter
                counter += 1
        return 'Index does not exist!'
```

```
def list_to_matrix(x):
    counter = 0
    for i in range(3):
        for j in range(3):
            if counter == x:
                return i, j
            counter += 1
    return 'Index does not exist!'

def findBlank_8p(state):
    ctr = 0
    for i in state:
        if i == 0:
            return list_to_matrix(ctr)
        ctr += 1
    return 'Blank not found!'

def swap(state, x1, y1, x2, y2):
    temp = state[matrix_to_list(x1, y1)]
    state[matrix_to_list(x1, y1)] = state[matrix_to_list(x2, y2)]
    state[matrix_to_list(x2, y2)] = temp

def actionsF(state):
    blank = findBlank_8p(state)
    validActions = []
    if blank[1] != 0:
        validActions.append('left')
    if blank[1] != 2:
        validActions.append('right')
    if blank[0] != 0:
        validActions.append('up')
    if blank[0] != 2:
        validActions.append('down')
    return validActions

import copy
def takeActionF(state, action):
    blank = findBlank_8p(state)
    state2 = copy.copy(state)
    if action == 'left':
        swap(state2, blank[0], blank[1], blank[0], blank[1] - 1)
    if action == 'right':
        swap(state2, blank[0], blank[1], blank[0], blank[1] + 1)
    if action == 'up':
        swap(state2, blank[0], blank[1], blank[0] - 1, blank[1])
    if action == 'down':
        swap(state2, blank[0], blank[1], blank[0] + 1, blank[1])
    return state2

def depthLimitedSearch(state, goalState, actionsF, takeActionF, depthLimit):
    if state == goalState:
        return []
    if depthLimit == 0:
        return 'cutoff'
    cutoffOccurred = False
```

```
        for action in actionsF(state):
            childState = takeActionF(state, action)
            result = depthLimitedSearch(childState, goalState, actionsF, takeActionF, 
            if result == 'cutoff':
                cutoffOccurred = True
            elif result != 'failure':
                result.insert(0, childState)
                return result
        if cutoffOccurred:
            return 'cutoff'
        else:
            return 'failure'

    def iterativeDeepeningSearch(startState, goalState, actionsF, takeActionF, maxDept
        for depth in range(maxDepth):
            result = depthLimitedSearch(startState, goalState, actionsF, takeActionF, 
            if result == 'failure':
                return 'failure'
            if result != 'cutoff':
                result.insert(0, startState)
                return result
        return 'cutoff'

    if __name__ == '__main__':
        state = [1, 0, 3, 4, 2, 6, 7, 5, 8]
        goalState = [1, 2, 3, 4, 5, 6, 7, 8, 0]
        printState_8p(state)

    # Bi-directional Search

    class adjacent_Node:

        def __init__(self, v):
            self.vertex = v
            self.next = None

    class bidirectional_Search:

        def __init__(self, vertices):
            self.vertices = vertices
            self.graph = [None] * self.vertices
            self.source_queue = list()
            self.last_node_queue = list()
            self.source_visited = [False] * self.vertices
            self.last_node_visited = [False] * self.vertices
            self.source_parent = [None] * self.vertices
            self.last_node_parent = [None] * self.vertices

        def AddEdge(self, source, last_node):
            node = adjacent_Node(last_node)
            node.next = self.graph[source]
            self.graph[source] = node
            node = adjacent_Node(source)
            node.next = self.graph[last_node]
```

```
            self.graph[last_node] = node

    def breadth_fs(self, direction = 'forward'):
        if direction == 'forward':
            current = self.source_queue.pop(0)
            connected_node = self.graph[current]
            while connected_node:
                vertex = connected_node.vertex
                if not self.source_visited[vertex]:
                    self.source_queue.append(vertex)
                    self.source_visited[vertex] = True
                    self.source_parent[vertex] = current
                connected_node = connected_node.next
        else:
            current = self.last_node_queue.pop(0)
            connected_node = self.graph[current]
            while connected_node:
                vertex = connected_node.vertex
                if not self.last_node_visited[vertex]:
                    self.last_node_queue.append(vertex)
                    self.last_node_visited[vertex] = True
                    self.last_node_parent[vertex] = current
                connected_node = connected_node.next

    def is_intersecting(self):
        for i in range(self.vertices):
            if (self.source_visited[i] and
                self.last_node_visited[i]):
                return i
        return -1

    def path_st(self, intersecting_node, source, last_node):
        path = list()
        path.append(intersecting_node)
        i = intersecting_node
        while i != source:
            path.append(self.source_parent[i])
            i = self.source_parent[i]
        path = path[::-1]
        i = intersecting_node
        while i != last_node:
            path.append(self.last_node_parent[i])
            i = self.last_node_parent[i]
        print("Path : ")
        path = list(map(str, path))
        print(' '.join(path))

    def bidirectional_search(self, source, last_node):
        self.source_queue.append(source)
        self.source_visited[source] = True
        self.source_parent[source] = -1
        self.last_node_queue.append(last_node)
        self.last_node_visited[last_node] = True
        self.last_node_parent[last_node] = -1
        while self.source_queue and self.last_node_queue:
```

```
            self.breadth_fs(direction = 'forward')
            self.breadth_fs(direction = 'backward')
            intersecting_node = self.is_intersecting()
            if intersecting_node != -1:
                print("Path exists between {} and {}".format(source, last_node))
                print("Intersection at : {}".format(intersecting_node))
                self.path_st(intersecting_node,
                                source, last_node)
                exit(0)
        return -1

if __name__ == '__main__':
    n = 17
    source = 1
    last_node = 16

    my_Graph = bidirectional_Search(n)
    my_Graph.AddEdge(1, 4)
    my_Graph.AddEdge(2, 4)
    my_Graph.AddEdge(3, 6)
    my_Graph.AddEdge(5, 6)
    my_Graph.AddEdge(4, 8)
    my_Graph.AddEdge(6, 8)
    my_Graph.AddEdge(8, 9)
    my_Graph.AddEdge(9, 10)
    my_Graph.AddEdge(10, 11)
    my_Graph.AddEdge(11, 13)
    my_Graph.AddEdge(11, 14)
    my_Graph.AddEdge(10, 12)
    my_Graph.AddEdge(12, 15)
    my_Graph.AddEdge(12, 16)

    out = my_Graph.bidirectional_search(source, last_node)

    if out == -1:
        print("No path between {} and {}".format(source, last_node))
```